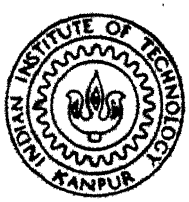
UNIX EMULATION SERVICES FOR PAWAN (I

By

Jean Jeeva Hazel Das



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1993

UNIX EMULATION SERVICES FOR PAWAN - I

A thesis submitted
in partial fulfillment
of the requirements
for the degree of

Master of Technology

by

JEAN JEEVA HAZEL DAS

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

April, 1993

To My Parents - Two Wonderful People

CSE-1993-M-IDAS-UNI

Acknowledgements

I express my sincerest thanks and gratitude to my guide, Dr Gautam Barua, who has been an inestimable help through the course of my thesis. His constructive comments and unerring instinct in pointing out the weaknesses in my arguments Helped me to form a clearer understanding and chart out a better course of action. I am greatly indebted to him for enriching my knowledge.

I thank Deepanker Bairagi, for being a very supportive thesis partner. It was with his help and co-operation that I was able to finish this thesis.

If it was not for my father's encouragement and support during my childhood, my life would have followed a different path altogether. My mother, with her great courage and love, enabled me to reach upto this level. Both my sisters with their love, laughter and sane advice, helped me view life in the proper perspective.

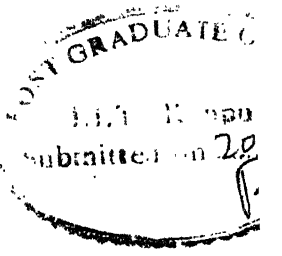
When things seemed too tough to handle, George was a rock of support, boosting up my morale and urging me on to greater efforts.

Sobha, Vimala, Ligy and Viji gave a jest to life with the lively discussions we used to have, all of us equally adamant about our views. Their teasing and friendship made the hostel a home away from home. I thank those wonderful relatives and friends who kept writing to me and praying for me.

I thank all my teachers, they have played an important role in shaping my future. I'll never be able to forget the care and concern they have shown.

J. J. HAZEL DAS

CERTIFICATE



This is to certify that the work contained in the thesis titled, **UNIX Emulation Services for PAWAN (I)**, was carried out under my supervision by **Jean Jeeva Hazel Das** and it has not been submitted elsewhere for a degree.

Gautam Barua
Professor
Dept. of Comp. Sc. and Engg.
I.I.T., Kanpur April 1993

Abstract

In this thesis the design and implementation of a UNIX emulation service for PAWAN is described. The trend in operating systems research in the last few years has been shifted from kernalization to dekernalization, i.e kernels enriched and overloaded with functionality and abstractions are steadily making way for micro-kernels. The design and implementation of PAWAN is based on MACH, a multiprocessor based micro-kernel developed at Carnegie Mellon University. The MACH micro-kernel was ported onto the M68020 based Horizon III and a set of user state servers was added to it by a group of four students. This thesis is a continuation of their work.

PAWAN consists of an emulation library and a set of user state servers which successfully exploit the MACH features to emulate UNIX. The user state servers provide file service, signalling facility, terminal handling, naming, process control and program execution. Currently, complete UNIX source code compatibilty has been achieved. By porting sophisticated utilities with minimal effort, a fullfledged operating system can be built on the top of it.

Contents

List of Figures

Chapter 1

Introduction

The enhancements in hardware technology, experience with existing systems, development of new applications and needs have resulted in dramatic changes in operating systems. The user has become literally swamped with systems offering newer and better facilities. Under the weight of changing needs and technologies, operating systems have been modified to provide a staggering number of different mechanisms for management of objects and resources. These new features have mostly been incorporated into the kernel, making it unwieldy and reducing modifiability.

The advent of high speed networks and the demand for integrated distributed systems has coincided with the development of better structures for conventional operating systems, allowing them to be built as a collection of intercommunicating software modules or processes each performing a well-defined function, such as file management, resource allocation or task scheduling. The advent of easily configurable open systems to which new services can be added without having to rebuild or restart existing system software, has been a positive step in this direction. In such systems the boundary between the operating systems and the programs developed by users is less rigid, allowing the system to be viewed as an extensible set of software resources and services. Examples of such systems are the MACH, AMOEBA, V-system, CHORUS and the LOCUS distributed systems.

The MACH technology has been designed for building the "new generation" of open, distributed scalable operating systems. The MACH kernel abstractions essentially provides a base upon which complete system environments may be built. At IIT Kanpur, the MACH microkernel has been ported to run on a network of MC68020 based uniprocessors and several user state servers has been implemented [Ashi92, Baru92, Gopa92, Rao92] in order to handle most of the functionality of the UNIX system. This and its companion thesis [Bair93] describes the enhancements to these services and the construction of an emulation library to provide complete UNIX source code compatibility.

1.1 MACH Operating System

The MACH micro-kernel developed at Carnegie Mellon University is a successor to the ACCENT project and is intended as a basis for development of distributed systems that include multiprocessors. It is an open system based on a lightweight kernel running in each computer with services such as file system, network services and process management outside the kernel [Coul88].

MACH is fundamentally a message passing communication kernel. Operations on objects other than messages are performed by sending messages to ports representing these objects. In this way MACH permits system services and resources to be managed by user-state tasks.

The model provided by MACH is a service model in which objects are managed by servers and clients make requests for operation on objects by using remote procedure calls. The Matchmaker remote procedure call interface language [Jone86] enables distributed programs to be built in several existing programming languages including C, Pascal, Ada and Lisp. Remote procedure calling in Matchmaker is supported by efficient and flexible interprocess communication facilities in the kernel.

MACH objects are represented by ports, which are protected message queues, and whose names can be transmitted in messages. By using Matchmaker, application programmers can be shielded from the intricacies of message composition, sending and reception and instead can be offered a procedural interface for sets of typed operations upon objects.

The MACH virtual memory management system exhibits architecture independence, multiprocessor and distributed system support and advanced functionality.

1.2 Overview of Other Distributed Systems

1.2.1 The V Distributed System

The V distributed system [Cher84], is an operating system designed for a cluster of computer workstations connected by a high performance network. The system is structured as a relatively small "distributed kernel", a set of service modules, various run-time libraries and a set commands. The kernel is distributed in that a separate copy of the kernel executes on each participating network node, yet separate copies cooperate to provide a single system abstraction of processes in address spaces communicating using a base set of communication primitives.

The existence of multiple machines and network interconnections is largely transparent at process level. The service modules implement value-added

services using the basic access to hardware resources provided by the kernel. The various run-time libraries implement conventional language or application-to-operating system interfaces.

The kernel is a software backplane that provides a base for building and configuring systems. It consists of lightweight processes and interprocess communication. New services are 'plugged in' and communicate with other processes using the IPC provided in the kernel and their internal design is invisible to the rest of the system. The V-kernel runs in each workstation and is based on clusters (V-teams) of threads, that can be dynamically created and destroyed and can share an address space. Processes communicate via shared memory within a V-team and via network transparent, synchronous message passing between different V-teams.

1.2.2 AMOEBA

The AMOEBA architecture consists of four principal components [Rene89], [Mull85], [Coul88]. First are the workstations, one per user, which run window management software and on which users can carry out tasks requiring fast interactive response. Then we have the pool processors, a group of CPUs that can be dynamically allocated as needed, used and then returned to the pool. The specialised servers such as directory, file server and various other servers provide specialised functions. Finally we have the wide area network gateways, which are used to link AMOEBA systems at different sites in possibly different countries into a single uniform system.

AMOEBA is an object oriented distributed system, the objects are abstract data types such as files, directories and processes and are managed by server processes. A client process carries out operations on an object (transaction), by sending a request message to the server process that manages the object. While the client blocks, the server performs the requested operation on the object and sends a reply message back to the client which unblocks the client. To handle multiple transactions going on at the same time a process can be subdivided lightweight subprocesses called threads. By having a thread for each request, a server process can handle multiple requests simultaneously. A client process can perform several transactions at the same time by having a thread per transaction.

1.2.3 The CHORUS Distributed System

The CHORUS distributed system enables us to integrate various types of operating systems - from small real-time systems to general-purpose operating systems, in a single operation system [Abro89],[Coul88]. CHORUS is a communication based technology. Its minimal real-time kernel integrates distributed processing and communications at the lowest level.

CHORUS operating systems are built as sets of independent system servers, to which the kernel provides the basic services such as activity

scheduling, network transparent IPC, memory management and real-time event handling. The kernel can be scaled to exploit a wide range of hardware configurations such as small embedded board, multiprocessor workstations or high performance server. The memory management service has been designed as a well isolated component offering generic interfaces adapted to various hardware architectures and to various system needs. The secondary storage objects are managed outside the kernel, within independent servers.

An actor is the unit of distribution in the CHORUS system. It defines a protected address space supporting the execution of threads which share the address space of the actor.

The thread is the unit of execution in a CHORUS system, it is characterised by a context corresponding to the state of the processor. A thread is always tied to one and only one actor, which constitutes the execution environment of the thread. Within the actor many threads can be created and can run in parallel. Threads making up an actor can communicate and synchronise with any other thread, on any site. The CHORUS virtual memory has been designed as a set of basic tools suited for versatile implementation of various system policies.

1.2.4 The X-Kernel

The X-kernel is a experimental operating system for personal workstations that allow uniform access to resources throughout a nationwide internet: an interconnection of networks similar to the TCP/IP internet [Pete90]. Workstations are viewed as a portal through which users access both local and remote network resources. This is realised by providing an infrastructure that is general enough to support a wide variety of protocols, yet efficient enough that no protocol suffers a serious performance penalty.

The X-kernel supports a library of protocols, and it accesses different resources with different protocol combinations. Two user-level systems have been built on top of the X-kernel to give users an integrated and uniform interface to resources. These systems, a file system and a command interpretor hide the differences among the underlying protocol.

The X-kernel currently runs on the SUN-3 workstations and incorporates components that manage processes, memory and communication. Multiple address spaces are supported, multiple light-weight processes can execute in each address space and processes within an address space synchronise using kernel supported semaphore. A communication manager is provided, which offers an object oriented infrastructure for composing protocols and a collection of powerful tools for implementing tasks common to all protocols.

1.2.5 The LOCUS Distributed System

LOCUS is a UNIX-like distributed system [Walk83],[Coul88]. It supports transparent access to data through a network wide filesystem, permits au-

tomatic replication of storage, supports transparent distributed process execution, supplies a number of high reliability functions such as nested transactions, and is upward compatible with UNIX. The system provides a high degree of transparency concerning the location of files and some degree of transparency concerning the location of process execution.

The system appears to clients like one giant UNIX system, with all the computers playing both client and server roles and with UNIX file access, process creation and interprocess communication primitives implemented transparently across the network. LOCUS is a procedure based operating system, processes request system services by executing system calls, which trap to the kernel.

The LOCUS filesystem presents a single tree structured naming hierarchy to applications and users. LOCUS names are fully transparent; it is not possible from the name of the resource to discern its location in the network. LOCUS provides nested transactions in which all changes to a given file are atomic. To implement transactions, both original and changed data are kept at the storage site for the file in question, until a transaction is complete. LOCUS provides replication of files at the granularity of whole directories on the grounds that, if some node in a tree is inaccessible, then all the files below that node are inaccessible.

1.3 Motivation

Continuously evolving microprocessors and the broad spectrum of new software has made configurability and modifiability, even at the cost of a little inefficiency, important factors to look for in a new system. MACH provides powerful standard micro-kernel, which can eventually run different operating system applications side by side on the same workstation.

We selected MACH for our purpose from the other alternative systems described above. LOCUS tries to provide a distributed environment for UNIX, resulting in a closed system with with a very large kernel [Coul88]. A extensive protocol library has been implemented in X-kernel to provide uniform access to resources in a nationwide internet. All the others have implemented lightweight kernels as a basis for building distributed systems. MACH provides multiple tasks, with multiple threads of execution and integrates both multiprocessor functionality. It has the technology to deliver a high performance inter-process communication. MACH is also exploring the extension of memory from the local machine to a transparent network-wide service [Gold89]. MACH is available and supported as a product by a number of hardware vendors, including NEXT. It is the base technology for the OSF/1 operating system from the Open Software foundation . MACH has become the most widely used micro-kernel not only for UNIX based

operating systems but even as the core for the next generation of IBM's OS/2[1].

The source code for MACH3.0 micro-kernel had been ported to the MC68020 based mini, as the basis for a distributed OS to act as an workbench for experimental distributed operating systems. Various distributed services has been provided by building a number of user-state servers. We have augmented the user-state servers and built an emulator for UNIX on top of the MACH kernel.

In this thesis and the companion thesis [Bair93], we present an overview and discussion of the major techniques and experiments that characterise the design of the emulation system. Some of the relevant aspects include the combination of several independent servers to create a complete system, generic service interfaces relying on the emulation library as a interface translator, and the moving of portions of system state and processing from servers into an emulation library.

1.4 Outline of Thesis

The Key ingredients of the MACH kernel and its outstanding characteristics are dealt with in the next chapter. Chapter 3 explores the PAWAN design and describes the UNIX emulation setup. The design goals and implementation details of the Process server are given in chapter 4. We describe the PAWAN implementation of process execution in chapter 5. Chapter 6 outlines the emulation library setup. In chapter 7, we have explored the possible server extensions and improvements. Details of the File server can be got from [Bair93]. The other user state servers are explained in [Ashi92], [Baru92] [Gopa92], [Rao92].

[1] A comment which appeared in the oct 1992 UNIX World.

Chapter 2

MACH Overview

This chapter describes MACH and the motivations that led to its design. The kernel abstractions and the features of MACH are examined. The basic abstractions and functions ported to the IITK implementation are also briefly discussed.

2.1 The MACH Micro-Kernel Architecture

The recent trend in operating system development consists of structuring the operating system as a modular set of system servers, which sit on top of a minimal micro-kernel. Examples of such systems have been given in the first chapter (MACH, CHORUS, AMOEBA and V-system). The micro-kernel provides system servers with generic tools, limited to process scheduling and memory management functions, independent of a particular operation system environment, and a simple Inter-Process communication(IPC) facility that allows system servers to interact independently of where they are executed, in a multiprocessor, multicomputer, or network configuration [Gien91].

The MACH micro-kernel has been selected as the base for our emulation system for UNIX. The basic kernel services form a standard base supporting the implementation of functions specific to a particular operating system environment. These system specific functions can be configured into user-state servers managing the other physical and logical resources of a computer system such as files, devices and high level communication services. This increases the modularity, portability, scalability and "distributability" of the overall emulated system. We can build and design emulation systems for a range of targets that span from relatively simple systems like MS-DOS to considerably larger systems such as VMS [Dean90],[Juli92]. The details of our UNIX emulation system is given in chapter 3.

2.2 Design: An Extensible Kernel

The MACH kernel abstractions provide a base upon which complete system environments may be built [Acce86], [Baro88], [Rash86]. MACH provides for the manipulation of system resources through this small set of machine-independent abstractions and for the integration of memory management and communication functions [Dean 90]. The MACH abstractions were chosen not only because of their simplicity but also for performance reasons. Substantial performance benefits are gained by integrating virtual memory management and interprocess communication [Rash86], [Youn87]. The MACH kernel provides five basic abstractions.

- A *task* is an execution environment in which threads may run. It is the basic unit of resource allocation and includes a paged virtual address space and protected access to system resources such as processes and ports. It is the framework in which a number of threads carry out computations.
- A *thread* is the basic unit of execution. It consists of a basic processor state, an execution stack and a limited amount of per thread static storage. It shares all other memory and resources with the other threads executing in the same task. It can execute only in one task.
- A *port* is a communication channel - logically a queue of messages protected by the kernel. Ports are the reference objects of the MACH design. Only one task can receive messages from a port, but all tasks that have access to the port can send messages to the port.
- A *message* is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain pointers and type capabilities for ports.
- A *memory object* is a secondary storage object that is mapped into a task's virtual memory. These are commonly files managed by a file server.

The MACH kernel functions can be divided into the following categories.

- basic message primitives and support facilities.
- port and port set management facilities.
- task and thread creation and management facilities.
- virtual memory management functions.
- operation on memory objects.

The provision of these basic functions in the kernel, makes it possible to run varying system configurations on different classes of machines while providing a consistent interface to all resources. The actual system running on any particular machine thus becomes a function of its server rather than its kernel.

2.2.1 Tasks and Threads

MACH divides the UNIX process abstraction into two orthogonal abstractions : the *task* and the *thread*. MACH allows multiple threads to execute within a single task. On tightly coupled shared memory multiprocessors, multiple threads within the same task may execute in parallel. Thus an application can use the full parallelism available and still incur modest kernel overhead.

Operations on threads and tasks are invoked by sending a message to a port representing a task or thread. Threads may be created (within a specified task), destroyed, suspended and resumed. The suspend and resume operations, when applied to a task, affect all threads within the task.

Tasks are related to one another in a tree structure by task creation operation. Regions of virtual memory may be marked for future child tasks as either inheritable, read/write, copy- on-write, or neither.

Application parallelism can be achieved in MACH in three ways

- through creation of a single task with many threads of control, which are able to execute in a shared address space, using shared memory for communication and synchronization.
- through related creation of many tasks that share restricted regions of memory.
- through the creation of many tasks that communicate via messages.

2.2.2 Virtual Memory Management

The MACH virtual memory design allows tasks to

- allocate and deallocate regions of virtual memory.
- set protections on regions of virtual memory.
- specify the inheritance of regions of virtual memory.

It allows for both copy on write and read/write sharing of memory between tasks. Virtual memory related functions, such as pagein and pageout, may be performed by non-kernel tasks [Youn87]. MACH does not impose restrictions on what regions may be specified for these operations, except hat they be aligned on system page boundaries.

An important feature of MACH's virtual memory is the ability to handle page faults and pageout data requests outside the kernel. When virtual memory is created, special paging tasks may be specified to handle paging requests. Thus to implement memory mapped files, virtual memory is created with its pager specified as the file system. MACH provides some basic paging services in the kernel. Memory with no pages is automatically zero-filled.

MACH implements virtual memory by mapping process addresses onto memory objects which are represented as communication channels and accessed via messages. This increases the flexibility in memory management available to user programs and improves performance. The memory object can be created and serviced by a user-level data manager task. This gives MACH the ability to efficiently manage system services like network paging and file system support outside the kernel.

MACH's virtual memory implementation is split between the *machine-independent* and the *machine-dependent* sections. This contributes to portability and supports very large but sparsely populated virtual memory spaces. The MACH address map need only contain descriptors for those regions of virtual memory actually occupied by useful items. In addition to the normal demand paging of tasks, MACH virtual memory implementation allows portions of the kernel to be paged.

2.2.3 Interprocess Communication

The MACH interprocess communication is defined in terms of ports and messages and provides both location independence, security and data type tagging.

The basic transport abstraction provided by MACH is the *port*. A port is a protected kernel object onto which messages may be placed and from which messages may be removed. It may have any number of senders, but only one receiver. Access to port is granted by receiving a messagge containing a port capability (to either send or receive).

Interprocess interfaces, including the interface to MACH kernel, are defined using an interface definition language called matchmaker. Matchmaker allows a program to specify an interface between client and server. It then locates the specification and generates stubs to create a distributed program without worrying about details of sending messages or type conversion between various machines.

2.3 MACH Tools

2.3.1 MIG - the MACH interface generator

MIG is an interim implementation of a subset of the Matchmaker language that generates C and C++ remote procedure call interfaces for interprocess communication between MACH tasks [Drav89]. The MIG program automatically generates procedures in C to pack and send, or receive and unpack the IPC messages used to communicate between processes.

The user must provide a specification file defining parameters of both the message passing interface and the procedure call interface. MIG then generates three files:

- User Interface Module: This module is meant to be linked into the client program. It implements and exports procedures and functions to send and receive the appropriate messages to and from the server.
- User Header Module: This module is meant to be included in the client code to define the types and routines needed at compilation time.
- Server Interface Module: This module is linked into the server process. It extracts the input parameters from an IPC message, and calls a server procedure to perform the operation. When the server procedure or function returns, the generated interface module gathers the output parameters and correctly formats a reply message.

MIG is implemented as a cover program that recognizes a few switches and then calls cpp to process comments and preprocessor macros such as #include or #define. The output from cpp is then passed to the program migcom which generates the C files. The server procedures must be written by the user. The server waits for procedure call invocations on its communication port. To use the calls exported by the user interface module a client must first access the port to call the server on.

2.3.2 C Threads

The C Threads package allows parallel programming in C under the MACH operating system [Coop88], [Golu87]. The MACH kernel does not enforce a synchronisation model, it just provides basic primitives upon which different models of synchronisation may be built. The C-threads provides a high level C interface to the low level thread primitives along with a collection of other mechanisms useful in various parallel programming paradigms. It provides

- multiple threads of control for parallelism.
- shared variables.
- mutual exclusion for critical sections.

- condition variables for synchronization of threads.

The C-threads package in MACH has three possible implementations. The first implements threads as coroutines in a single task, the second uses a separate task for each C-thread, using inherited shared memory to partially simulate the environment in which multiple threads run. The third implements C-threads using MACH threads. IITK MACH provides the third implementation.

2.4 MACH I/O Structure

The MACH I/O configuration is different from that of the current systems. The peripheral devices are accessed through the device server, which is implemented as a kernel object. Operations on the device server are invoked through request messages sent to the port representing the device server.

We have three device drivers, the terminal driver, the disk driver and the network driver.Device switch `dev_name_list` describes the devices possibly attached to the system. The table entries describes the entry points to the driver. We have yet another table, which gives the bus specification of devices. Character and block devices reside on the same table. MACH devices are accessed through block and character device interfaces to the kernel.

2.4.1 I/O Processing

To perform I/O on a device, a device_open request is made to the device server, which allocates a port for the device. This port can be used to perform I/O on that device. We can have a trusted server performing access mediation for the device, but handing out the port to various tasks at its discretion. Send rights to the device server port implies complete control over all the devices, whereas access to a device port imply control over that particular device.

Normal I/O is done by sending request messages to the device port. I/O can be inband or out of band as suitable for specific devices. Data is not buffered by the device server to avoid hardwiring the policies within it. The applications which uses its services are expected to employ their own buffering schemes.

A special entry point is provided in the device switch to handle asynchronous input. This point `device_set_filter`, associates a filter with a port. The input passes through this filter before being queued at the specified port. This facility is in fact used by the TTY server to handle special characters.

2.5 The Kernel File System and Network Support

The MACH kernel provides a basic file system, the boot_ufs (derived from the Berkeley Fast File system) within the kernel. It is made up of blocks of at most 8K units, with the smaller units (fragments) only in the last direct block. The fragment sizes are multiples of the device block. The inode is the focus of all file activity in the system. A unique inode is allocated for each active file, current directory , mount files, text file and the root.

By itself, the Mach kernel does not provide any mechanisms to support inter-process communication over the network. However, the definition of Mach IPC allows for communication to be transparently extended by user-level tasks called Network Servers. The BSD UNIX socket facility for network communication [Leff89], has been implemented inside the IITK MACH kernel [Baru91]. The user state network server can be written over this socket facility.

Chapter 3

The Design of PAWAN

In this chapter, we discuss the design goals of PAWAN, the emulation services, the user state servers in PAWAN and some issues regarding integration of the servers, user interface, authentication and system initialization etc.

3.1 Design Goals

Defining and standardizing a powerful micro-kernel base is only the first step in realizing the potential of the overall MACH approach. The next step, and perhaps the one richest in design possibilities, is to learn how to construct a wide range of useful higher-level systems on the top of this simple kernel. A particular class of such systems is the so-called emulation systems, that implement the application programming interface of an existing complete operating system or target system with various combinations of user level components viz. servers and/or libraries operating on the top of a MACH kernel. The main benifits expected from this overall emuation approach include increased modularity, portability, flexibility, security and extensibility, as well as simpler development, debugging and maintenance.[Juli92]

A major goal of PAWAN is to implement UNIX as an application program or to be specific, as a user task. Beyond the obvious advantages common to all client/server models, treating Unix as an application program has a number of implications :

- tailorability : versions of UNIX such as 4.3 BSD, System V.4 can be treated as different applications which can be potentially run side by side.
- portability : A considerable portion of Unix code is not machine dependent.
- extensibility : New versions of Unix can be implemented or tested alongside existing versions.

- real-time : traditional barriers to real-time support in UNIX can be removed both because the kernel itself does not have to hold interrupt locks for a long period of time to accomodate UNIX services and because the UNIX services themselves are preemptable. [Dean90]

There are two basic approaches to this design :

1) Multi-threaded User-state Unix server: Such a design is shown in Fig 3.1 This idea has been implemented in CMU. A single server, implemented as a MACH task with multiple threads of control managed by the C threads package, provides the UNIX semantics. Its key components are :

- A transparent UNIX support library:
  - Interprets all UNIX traps
  - Some simple traps are handled locally
  - Other traps are translated into messages to server.
- Unix Server:
  - Derived from original BSD implementation
  - pageable, interruptible, multithreaded
  - Acts as memory object pager for Unix inodes

2) Multi-server user state Unix : The key components of such a system are:

- A transparent UNIX support library:
  - Interprets all UNIX traps
  - Some simple traps are handled locally
  - Other traps are translated into messages to server.
- Generic (non-UNIX) servers:
  - Name server, Authentication server etc.
- User specific servers
  - File server, pipe server etc.

Another approach might be a compromise between the above two i.e. having one main central server and a number of auxiliary servers implementing specific portions of functionality.

PAWAN has an emulation library and several user state servers which provide UNIX services in a completely transparent manner. The choice of this design has been largely determined by extrapolating from the architecture of the native implementation of the target operating system UNIX. In

a shared memory multiprocessor machine, a single monolithic UNIX server implemented through system call and exception redirection might prove to be more efficient [Acce86], [Teva87d], [Youn87]. But in a loosely coupled LAN based environment as ours, a centralized UNIX server which services every user request, system call or exception in the network will clearly be extremely inefficient [Ashi92], [Baru92], [Gopa92], [Rao92]. For reasons stated above, the second approach has been chosen.

A major issue involved here is the separation of the service layer to two separate layers, one for generic and one for target specific functions. The target specific layer can be typically an emulation library. Generic servers can provide union of services provided by many different target systems. But generic interfaces are prohibitively complex and expensive although such a design will facilitate emulation of more than one operating system. Therefore, at the most one may afford to have only a partially generic service layer. The major driving factor behind the design of PAWAN is UNIX emulation and more particularly UNIX source code compatibility.

One might point out that in such systems, each individual component, i.e. a server or a library is typically designed specifically for the particular system in which it is to be used, and implemented either from scratch or by adapting code from the target OS implementation. Therefore, a major goal has been to make use of the existing unix code wherever possible (File server and Network server).

In order to maximize the overall flexibility, it is desirable to have as many independent servers as possible. These servers can then be used as a collection of standardized building blocks that can be assembled in various ways to create different systems. In addition to flexibility, such an architecture also increases the security and robustness of the system by isolating faulty or potentially malicious components into separate protected address spaces. Moreover it sometimes simplifies the implementation of some servers. However, such a desire for maximum modularity must be balanced against a number of practical considerations [Juli92]. For example, interaction between servers are more expensive than interaction between modules inside a particular server. The separation of services and corresponding interfaces must be carefully defined to minimize those interactions.

Keeping these issues in view, UNIX services in PAWAN has been divided into six user-state servers, each of which is accompanied by a library. The servers are Exec server, File server, Process server, Environment server, TTY server and Network server. The overall design of pawan is given in Fig 3.2. Each emulated process is a separate MACH task that contains the unmodified user code from various application programs and an emulation library that intercepts and implements system calls issued by instructions embedded in the user code. A brief description of the servers is given in the next section.

3.2 Description of the Servers

3.2.1 The Process Server

The full functionality of the UNIX signalling facility is provided by PAWAN. The Process server acts as the forwarding agent for all the *kill* requests. We go for a server based implementation since we need access to the task kernel ports. The SIGKILL type of signals which can't be blocked, caught or ignored are taken care of by the Process server. A task which handles signals will have a thread waiting for signals on its exception port. Since the Process server contains entries by all the user task, the default action (stop, terminate or resume user task) can be done by the server.

The Authentication server is implemented as a privileged thread within the Process server. This turns out to be quite efficient since the Process server tables contains information about all user processes(tasks).

3.2.2 The Exec Server

PAWAN provides UNIX style *execve* as well as *run* (a new call, details in chap.5). These are implemented as emulation library routines. However the implementation of setid exec is not possible at this level. We need to update the privileges ,and this is possible only by the trusted servers.

We run the Exec server with root privileges at startup time to take care of the setid exec. The *execve* and *run* library routines detect the need and contacts the Exec server when a setid exec or run comes up. The Exec Server then takes up the responsibility of updating the privileges and setting up the process.

3.2.3 The File Server

The File server in PAWAN is completely compatible with 4.3 BSD. It provides UNIX-like file I/O and can be integrated with the Network server to provide transparent network-wide file access and shared memory [Rao92]. It has been implemented by porting the 4.3 BSD file system code and uses the Device server inside the kernel for device I/O. The major difference of the File server with the BSD approach is the maintenance of U area inside the File server itself. It uses the notion of threads to exploit concurrency and uses features like copy-on-write to minimize copies of data.

3.2.4 The Environment Server

The Environment server facilitates the sharing of named variables between tasks. It is an extension of the MACH EnvManager. An environment is a set of named variables which can be read or changed via calls on an Environment Port. Variables can be strings, ports or environments. An

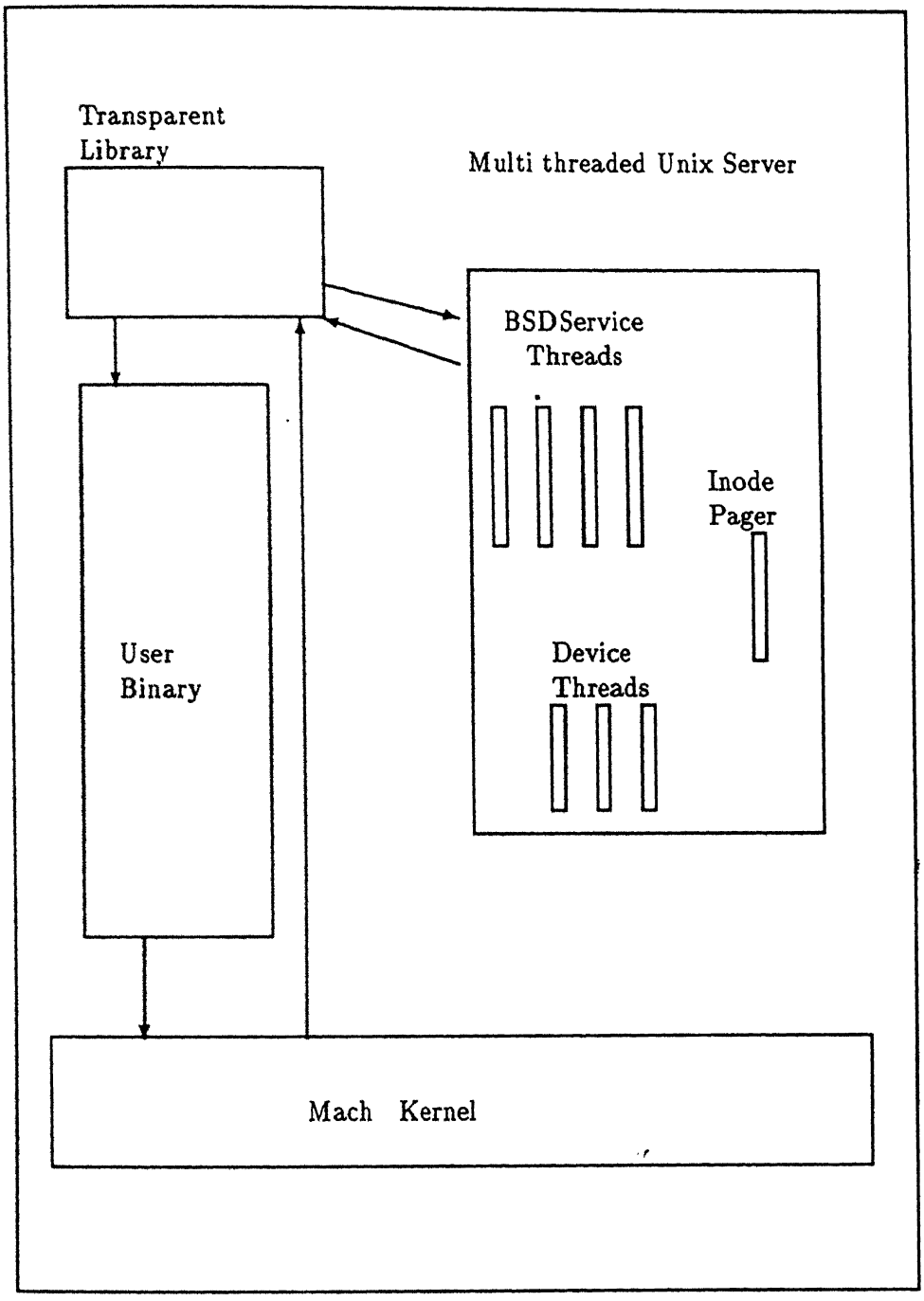


Figure 3.1: The Single Server Model

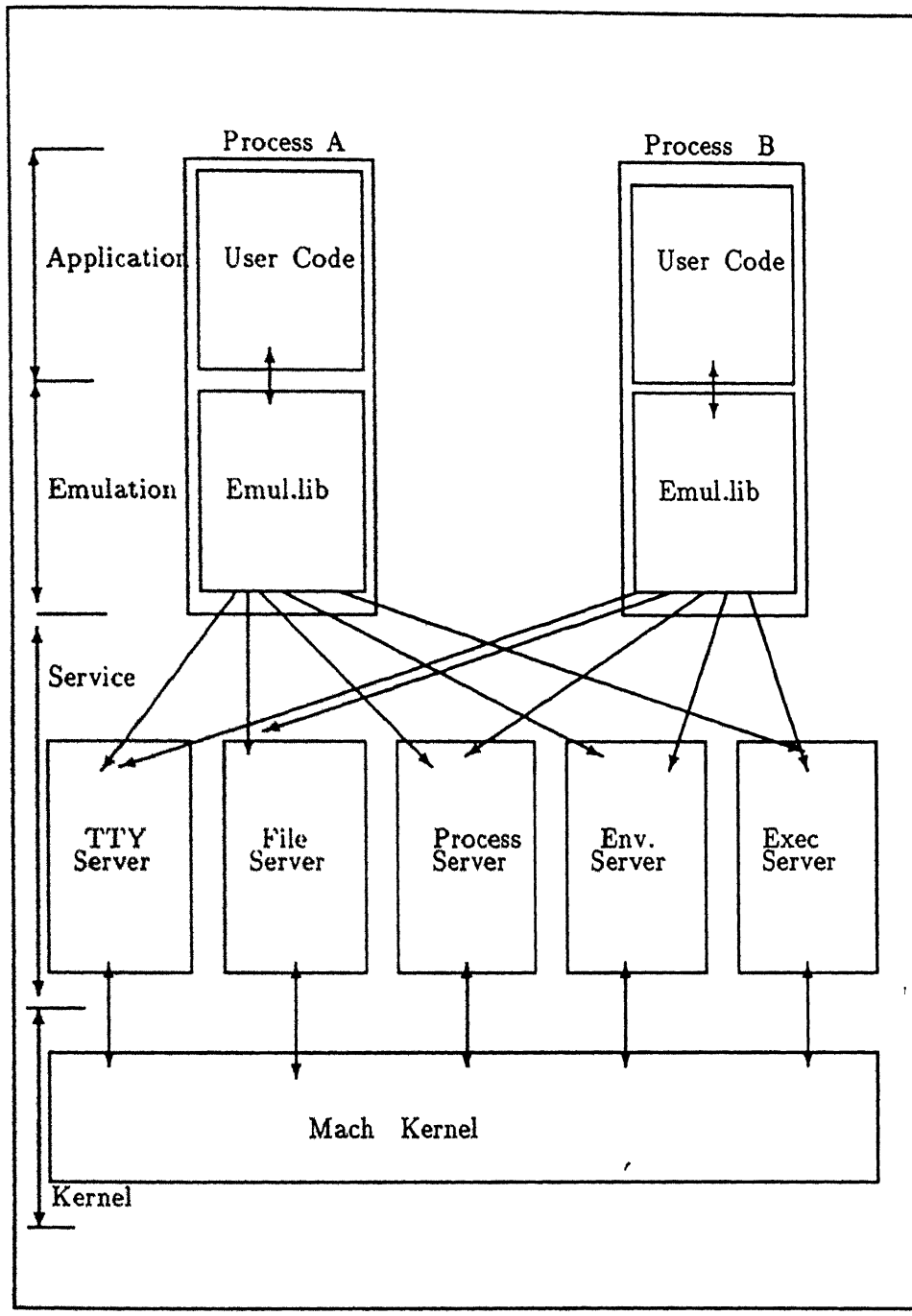


Figure 3.2: The design of PAWAN

environment may be shared between parent or child tasks, or an environment can be copied, and the copy can be passed to a child task. It is also possible to get a read-only port to an environment which allows reading but not modification of the environment. Variables stored in an environment are accessible through a specific server port. There may be one read/write and one read-only port to the same environment [Thom87]. Default or empty environments can be created with system wide constants useful to all tasks (like the hostname, network-address and well known server ports) and can be copied to one another. Ports and strings can be registered in or looked up from an environment by the tasks sharing it. A task can also use more than one environment: it could have access to a widely shared "global" environment as well as its own local environment. These features play an important role in system initialization and authentication. These issues are discussed later in this chapter. For detailed description of the environment server please refer to [Gopa92].

3.2.5 The TTY Server

The TTY server regulates access to terminals and has been designed to serve any user interface in general. Although in UNIX all device related services are put inside the file system, there are certain fundamental differences between user interface devices and a disk which make it advantageous to separate the code handling user-interface from the file system. It also provides facilities for job control, process groups and special character interpretation through signaling. In order to do so, the terminal driver was modified to use a special filtering mechanism provided in the MACH device server. To facilitate I/O redirection, the interface provided by the terminal and file servers for similar operations like open, read, write, close etc have been kept as close as possible.[Gopa92]. The emulation library uses this facility to successfully implement I/O redirection.

3.2.6 The Network Server

The Network server provides a socket based connection oriented and datagram services using the IP suit of protocols. It runs as as a privileged kernel task with user interface through system calls. By itself, the MACH kernel does not provide any mechanism to support inter-process communication over the network. However, the definition of MACH IPC allows for interprocess communication to be transparently extended by user level Network servers. They can act as a local representative for tasks on remote nodes. Messages destined for ports with remote receivers are sent to the local network server which forward it to the destination network server which, in turn, delivers it to the local receiver task. These user state network servers can be written over the socket facility provided by the kernel network server [Baru92].

Of all the servers mentioned above, the environment server and the TTY server belong to the genre of generic or partially generic servers. On the other hand, the other servers are mostly designed according to the UNIX semantics. The Authentication server is implemented as a privileged thread of the process server. It may be implemented as a separate independent server. A possible extension to this set is a pipe server.

All the servers in PAWAN are statefull. They maintain some state for every task accessing the server. This becomes necessary because the users are not trusted and if no state is maintained, the server has to believe in user information and that compromises security. The state includes usually the authentication information and the server state corresponding to that task e.g. ports for communication, data structures storing some information about the user etc.

3.3 The Emulation Library

Besides doing "emulation" and thereby providing an interface to the servers, the emulation library manages a considerable portion of process state. Thus, it displaces some of the processing required to implement various functions from the servers into the clients of these services themselves, and helps in concentrating system state in these clients. For example, it manages important pieces of information like the UNIX file descriptor table, signal mask etc. Although the user code and the emulation library code reside in the same address space, one may use the terms "executing in user space" and "executing in emulation space" to separately indicate the execution of the library routines [Juli92]. The emulation library is discussed in detail in Chapter 6.

3.4 General Issues

In this section we discuss some of the general issues regarding system initialization, authentication, user interface etc.

3.4.1 Public and Private Names

The concept of public and private names is implémented by the environment server. A public name is accessible to all tasks but a private name is accessible only to some chosen tasks which, in PAWAN, are the well known servers. When a user task looks up a public name, even if it is not present in its environment, it is looked up in the environments of the well known servers. On the other hand, when it looks up a private name, request is granted only if it happens to be a well known server itself.[Gopa92]

3.4.2 Server Interface to a User Task

The notion of public and private names introduced above is used to provide a secure user interface to servers. All servers have a public port and a private port which are registered in the Env server. A public port provides system wide services to all users. A private port is reserved for communication among the servers to carry out privileged opearations. The environment server's public port is available to all tasks. Any task that wishes to request some service from a server, initializes its environment and gets its own bootstrap port. Then it sets the bootstrap port to the Env server public port. It then obtains the public port of the server it wants to contact by requesting the Env server. After obtaining the public port of the required server, the user task directly talks to the server. The server usually returns a port, which communicates with the user task and serves all future requests through this port. Figure 3.3 describes the interaction between a user task and a server. The primary mode of communication in the system is the MACH IPC facility (MIG). It applies to communication between a user task and a server as well as communication among servers.

3.4.3 Identification of Tasks and Authentication

It follows naturally from the MACH philosophy that the identification of a user task is done by its kernel port. But unlike UNIX, where processes have a unique global pid, the task-id of a user-task is different in different servers since rights of a port are not unique to all tasks. But it does not pose a problem as long as they are appropriately translated and remain unique to a particular server while the server is alive.

Although for MACH applications it is possible to maintain security since ports are kernel protected capabilities, for UNIX applications UNIX-style authentications are necessary to provide protection against unauthorized and erroneous access to data. As mentioned above, the Authentication server is implemented as a thread of the Process server. It issues tokens of authentication and saves information about all the current tokens issued to users. Other servers ask the authentication server to verify whether an authentication token is valid. Whenever a new task comes up in the system its id is obtained from the File server or from its parent and a tuple is formed with the authentication token. This token is stored in a fixed memory location of the user task. The emulation library routines access this tuple and send it along with other information during each server call .

3.4.4 System Initialization

The Environment server is the first server to come up in the system. After the kernel startup, the Environment server is exec'ed by the bootload program. At first, it sets its bootstrap port to Env server public port. In the

next step, it creates five tasks for Exec server, File server, Network Server, process server and TTY server respectively. It then creates the environment for each server task created above and creates public and private ports for each server, and registers them in its environment. After this, it sets the bootstrap port of each server to their env_ports. It then executes the corresponding executable files on the five tasks mentioned above by using the bootload program.

The initialization procedure done at each server is as follows :

1. my-env_port = env_init(..)
2. my_public_port = env_get_port(..)
3. my_private_port = env_get_port("..Serverprivateport",..)
4. Get public and private ports of every other server and make state for them.
   make_server_state(servername, publicport, privateport);

The idea of make state is like initializing uareas for communication with other servers directly: they are supposed to use their private ports as the credential just as the ordinary user uses his kernel port.

After all the servers come up, TTY server which happens to be the last one to come up starts the login shell.

3.5 Login Shell

The login shell is spawned by the TTY server. It puts the login prompt on the terminal screen and waits for the user to type his login-name and then the password. it then checks the password by asking the file server. If the password matches, it prints the shell prompt on the screen and waits for the user to type his command. It then interprets the command and calls *run* to execute the command. After the command is executed, it prints the shell prompt on the screen and waits for the next command. This goes on till the user types "exit". A possible extension of this project may be to port a standard UNIX shell like the Bourne shell.

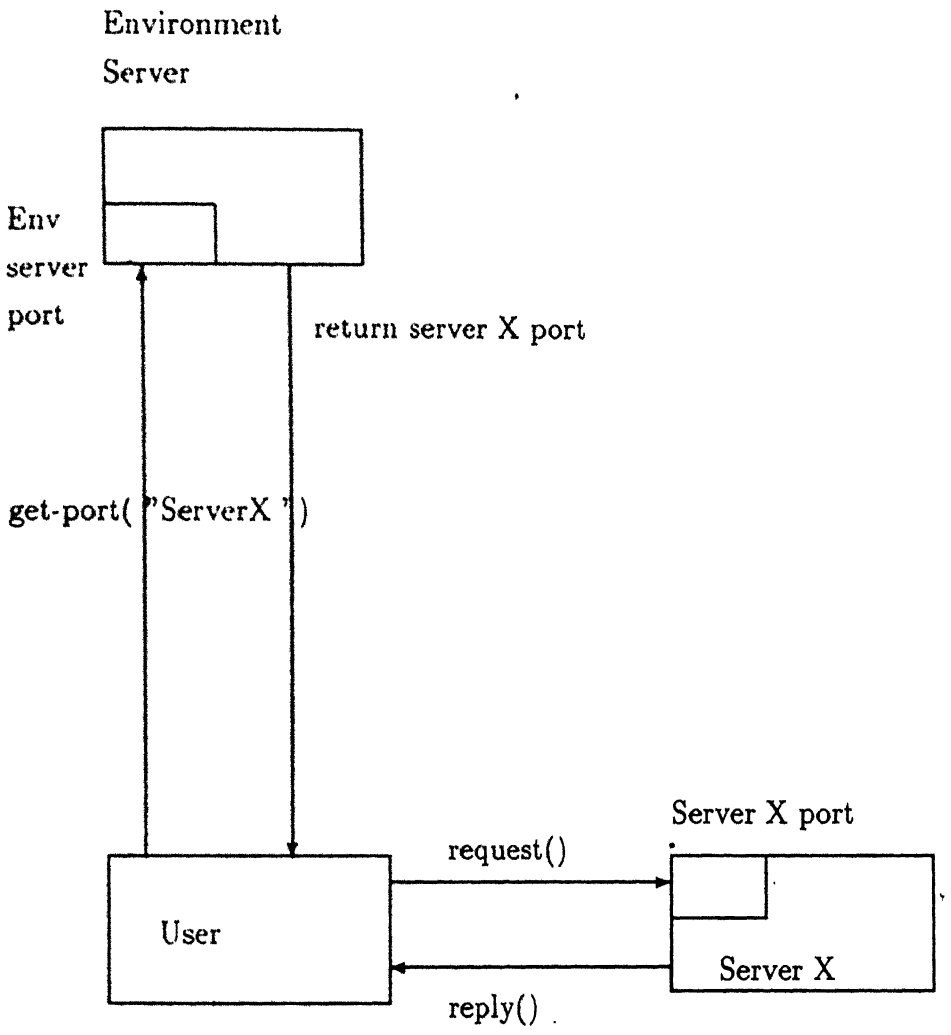


Figure 3.3: Interaction between a user and a server

Chapter 4

Process Server

4.1 Introduction

The Process server stores process related information and monitors all user processes. It assigns the unique process_id which specify the relation of processes to each other. The authentication server is implemented as a privileged thread within the process server.

UNIX kernel contains a number of services controlling execution of processes by allowing creation, termination or suspension of processes and communication between processes [Bach86]. We have gone for a server based implementation of the UNIX signalling facility. The implementation of process execution in PAWAN is explained in chapter 5.

The process server provides the necessary kernel support for the implementation of the UNIX signalling facility. The `wait` emulation library call is also implemented using the calls exported by the process server.

In the next section, we have described the design goals and implementation of the process server. This is followed by a brief description of the signalling mechanism in UNIX. The PAWAN implementation of signalling is explained in sec 4.4 The implementation of the authentication server is given in section 4.5.

4.2 Process Server

4.2.1 Design Goals

The process server was designed to provide the following functionality

- maintenance of process related variable (stored in the process table and uarea in UNIX).
- provide unique process identification.

- support for UNIX signalling and process synchronisation
- handle security issues.

At the relevant places, we have described why we have tried to provide the above functionality in the process server. The process server design is an extension with certain modifications of the Signal server design proposed by Ashish Singhai [Ashi92].

4.2.2 Process Server Design

There are two main threads of execution within the server. The Authentication server runs as a privileged thread waiting for authentication requests at the private port of the server. We have the request thread waiting for messages at the public port of the Process server.

We have seen that the MACH kernel provides just the basic primitives related to process scheduling and memory management functions. In order to provide comprehensive UNIX services through emulation, the process state must be stored somewhere. The emulation library manages some of the process related information. The remaining part of the process information is distributed among the various servers (for e.g. uarea information is stored by the File server [Bair93]).

The Process server stores a certain amount of process related information like process identifier (PID), event descriptor etc., in its server table. The maintenance of this information is done by the Process server when it receives process related calls at its private or public ports. The entries regarding a task is made by the Process server when it receives information about the existence of a new task (via the *task_create_msg* server call). The server assigns a unique pid to the task, associates the correct parent pid and duplicates the rest of the information from the parent task's entry. The Process server receives information about a task's changed state (from the *exitp* library routine or through the receipt of a signal). It then makes the relevant changes to the task state and sends a SIGCHLD signal to the parent task. The process id is used to identify user processes (used to implement *fork*, *wait* etc.).

The Process server contains information about all the tasks in the system. Information regarding process creation and destruction reach the Process server through the *task_create_msg* and *task_terminate_msg* calls exported by the server. Thus the process server tables glean information about all the user tasks, ensuring a reliable implementation of the signalling facility and enabling signalling between unrelated tasks. This also justifies our design decision to implement the Authentication server as a special thread within the Process server.

4.2.3 Process Server Calls

The process server exports the following functions to user processes (calls made on its request port)

- *task_create_msg*: Called by the login server when a new user logs in to the system, or by the *exec_run* or *fork* emulation library routines to register a new process.
- *task_terminate_msg*: Mainly used by the *exitp* emulation library routine to indicate the death of an emulated process.
- *kill_task*: The emulation library accesses this to implement the *kill* system call.
- *get_exit_status*: The *wait* system call is implemented using this. At the receipt of this call the server checks the process server table to access state of the child processes (if any). The status and child PID is sent back, if there are any zombie children.

Along with these calls, the process server provides the *task_to_pid*, *task_to_ppid*, *task_to_gid* and *task_to_uid* calls, used by the emulation library to implement the corresponding getid calls. Another call that has been implemented is the *reset_sigport* call used by the execve routine.

4.3 UNIX Signals

Though UNIX signals were initially intended for indicating exceptional events, with the introduction of job control, the signalling utility became more widely used. BSD 4.3 UNIX extended the prevalent signalling facility to improve reliability [Leff89]. Signals are software equivalents of traps or interrupts. The signal handling routines perform the equivalent function of interrupts or trap service routines. UNIX processes may send signals to each other with the *kill* system call, or the kernel may send signals internally. UNIX defines a set of signals for software and hardware conditions that may arise during the normal execution of a program.

UNIX signals can be classified under the following groups.

- Signals related to termination of a process.
- Signals related to process induced exceptions.
- Signals having to do with unrecoverable conditions during a system call (e.g. running out of system resources during *exec* after original address space has been released)
- Signals caused by unexpected error conditions during a system call (e.g. making a non existent system call).

- Signals originating from a process in user mode (e.g. signalling using *kill* system call).
- Signals related to terminal interaction.
- Signals for tracing the execution of a process.

Signals may be either delivered asynchronously to a process through application specified signal handlers or may result in default actions carried out by the system [Leff89], [Bach86]. The default action may be either of the following.

- Ignoring the signal.
- Terminating the process.
- Terminating the process after generating a core file that contains the process execution state at the time the signal was delivered.
- Stopping the process.

A signal handler is a user mode routine that the system will invoke when the signal is received by the process. The two signals SIGKILL and SIGSTOP, may not be ignored or caught. This ensures a software mechanism for stopping and killing runaway processes. The uarea contains an array of signal handler fields, one for each signal defined in the system. The kernel stores the address of the user function in the field that corresponds to the signal number.

The signals are posted to the system when a hardware event such as an illegal instruction is detected. This is also done when it detects a software event such as a stop request from the terminal. A user process can also send signals to another process using the *kill* system call. In this case the sending process is allowed only to send signals to processes with the same effective user identifier (unless the sender is superuser).

4.4 Signals in PAWAN

There are some factors which make signalling mechanism unnecessary in MACH based systems. The IPC mechanism can conveniently handle most of the functions performed by signals in UNIX. The blocking and non-blocking send and receive operations render signals unnecessary for synchronisation. The MACH kernel with the help of the exception server reports exceptions by sending messages to the task and thread exception ports.

In PAWAN, we have provided a UNIX style signalling facility with the help of the process server. This is in accordance with our design philosophy of providing UNIX services in a transparent manner (with provision to construct a wide range of other useful systems over the servers).

An important factor to keep in mind when implementing the UNIX signalling facility, is that runaway tasks should not be allowed to defy SIGKILL. This neccesitates having access to the task's kernel port. The default processing, for tasks that do not handle signals, should also be taken care of. The need for a reliable implementation as well as the need for some sort of kernel support (to implement SIGKILL and handle the default processing) induced us to go for a server based implementation of the UNIX signalling facility. Here we take advantage of the MACH kernel property which allows a task to make kernel calls on behalf of another task (should have acceess of the task kernel port). Thus we can make kernel calls requesting it to terminate, suspend or resume tasks. The kernel port of all emulated processes are sent to the process server via the *task_create_msg* during the task creation.

4.4.1 Process Server Support for Signalling

When the user process invokes the *kill* system call, the emulation library sends a kill request to the process server. The process server acts as a central co-ordinating agent, receiving kill requests and redirecting them to the specified user process (or task). It checks for permission and enforces proper handling of the SIGKILL and SIGSTOP signals. If the request passes through all the validity checks, then the signal is processed. A null signal port entry in the recipient task's signal table entry signifies that the process doesn't intend to catch signals, and hence the default action corresponding to the signal is taken [fig.4.1]. If a signal port is specified, the signal is sent to that port. The signal thread receives the signal and processes it.

If the function specified for the signal in the process is SIG_DFL, the reply message by the signal port sets a flag and the default processing is done by the process server [fig. 4.2]. The algorithm for *kill_task* is shown in [fig. 4.3].

4.5 Authentication Server

Any system should provide protection against unauthorized and erroneous access to data and against interference with the operation of the system. To implement security, protection methods are required providing facility for authenticating users and for controlling access to files, communication channels and other objects.

In UNIX, the kernel associates two user_ids with a process, independent of the process_id, the real user_id and the effective user_id [Bach86]. The real user_id identifies the user who is responsible for the running process. The effective user_id is used to assign ownership of newly created files, to check file access permissions, and to check permissions to send signals to processes via the *kill* system call. The kernel allows a process to change its

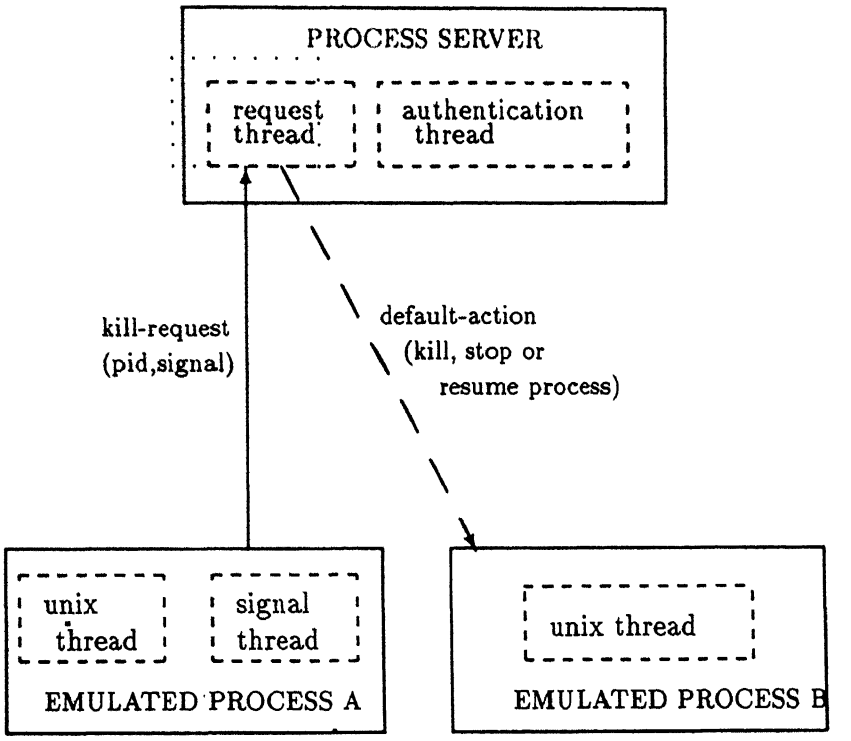


Figure 4.1: Signalling using *kill* (B doesn't handle signals)

effective user_id when it executes a *setuid* program or when it invokes the *setuid* call explicitely.

In PAWAN the state of the system is effectively distributed among the various servers and with a emulation library operating in each emulated process. The real and effective user_ids are stored in each of the trusted servers, which use it to authenticate the requests. This information is also stored by the emulation library.

The privileged calls (involving changing the server state, like changing access rights of a user) are allowed only on the private ports of the servers. Since these ports are accessible to other trusted servers, the data within the servers is protected against violation by the user.

The level of protection provided with just this mechanism is not enough to provide UNIX style authentication. The decisión to place more responsibility for various system functions in an emulation library that is not protected from incorrect or malicious user programs has implications in the area of robustness and security. A malicious client must be prevented from gaining access to protected resources.

The authentication server issues tokens of authentication and saves information about all the current tokens issued to users [fig. 4.4]. The other servers check in with the authentication server to see whether a authentica-

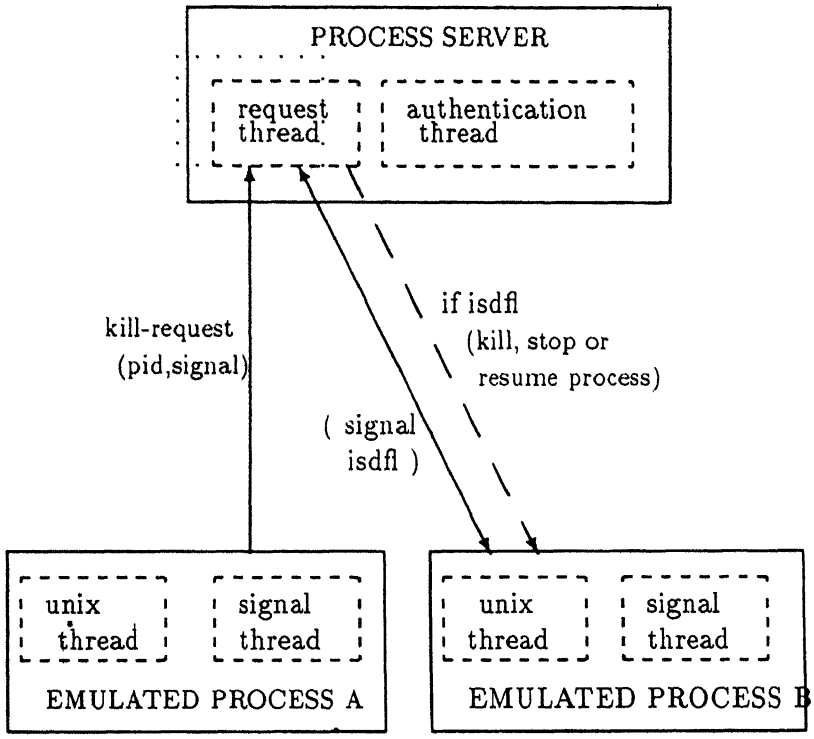


Figure 4.2: Signalling using kill (B handles signals)

tion token is valid [fig. 4.6]. The token is stored by the servers (for example the file server [Bair93]) and further authentication requests are made only if there is a mismatch between the cached value and the value contained in the service request sent to the server. We have defined a (task_id, token) tuple which is sent in the requests to enable servers to identify and authenticate the user task. The send rights of the kernel port of a user task is used as the task_id. This can be destroyed or even swapped, but never actually forged. The unique token is assigned by the authentication server when a new task is created. The token along with the user_id and group_id is stored in fixed memory locations of the user task. This is accessed by the emulation library routines and sent along with the task_id during each server call. This information is duplicated in all the child tasks associated with this task. A new token is assigned by the authentication server when a new set of credentials is to be installed for a task. This is usually done during a *setid* exec [fig. 4.5].

The authentication server requests can only be made on the private port of the process server. Thus the authentication information is accessible only to the trusted servers and the emulation library operating in each emulated process. This ensures that the authentication information is not forged or tampered with. The main calls exported by the authentication server are

```
kill_task(processserverport,killer_ent,killed_ent,sig);
mach_port_t processserverport;
sig_entry_t *killer_ent, *killed_ent;
int sig;
{
  if (invalid_signal_num(sig)
   return ERROR;
  if (unknown_entry(killer_ent) || unknown_entry(killed_ent))
   return ERROR;
  if (!superuser(killer_ent->task) &&
           have_authority(killer_ent->task,killed_ent->task))
   return ERROR;
  if (sig == 0)
   return 0;
  task_resume(killed_ent->task); /*wakeup if asleep*/
  if (null_sigport(killed_ent)
       ||cantcatch(sig))     /*check for SIGSTOP and SIGKILL*/
    local_default_action(killed_ent,sig);
  send_kill_msg(killed_ent->sigport,sig,&flag);
  if (flag)
    local_default_action(killed_ent,sig);
  return 0;
}

local_default_action(killed_ent,signal)
sig_entry_t killed_ent;
int         signal;
{
  switch(default_action(sig)) {
     case KILL_W_CORE : make_core();
     case KILL        : send_signal(killed_ent->ppid,SIGCHLD);
                        task_terminate(killed_ent->task);
                        dealloc_entry(killed_ent);
                        break;
     case STOP        : task_suspend(killed_ent->task);
    }
  return;
}
```

Figure 4.3: Algorithm for *kill_task*

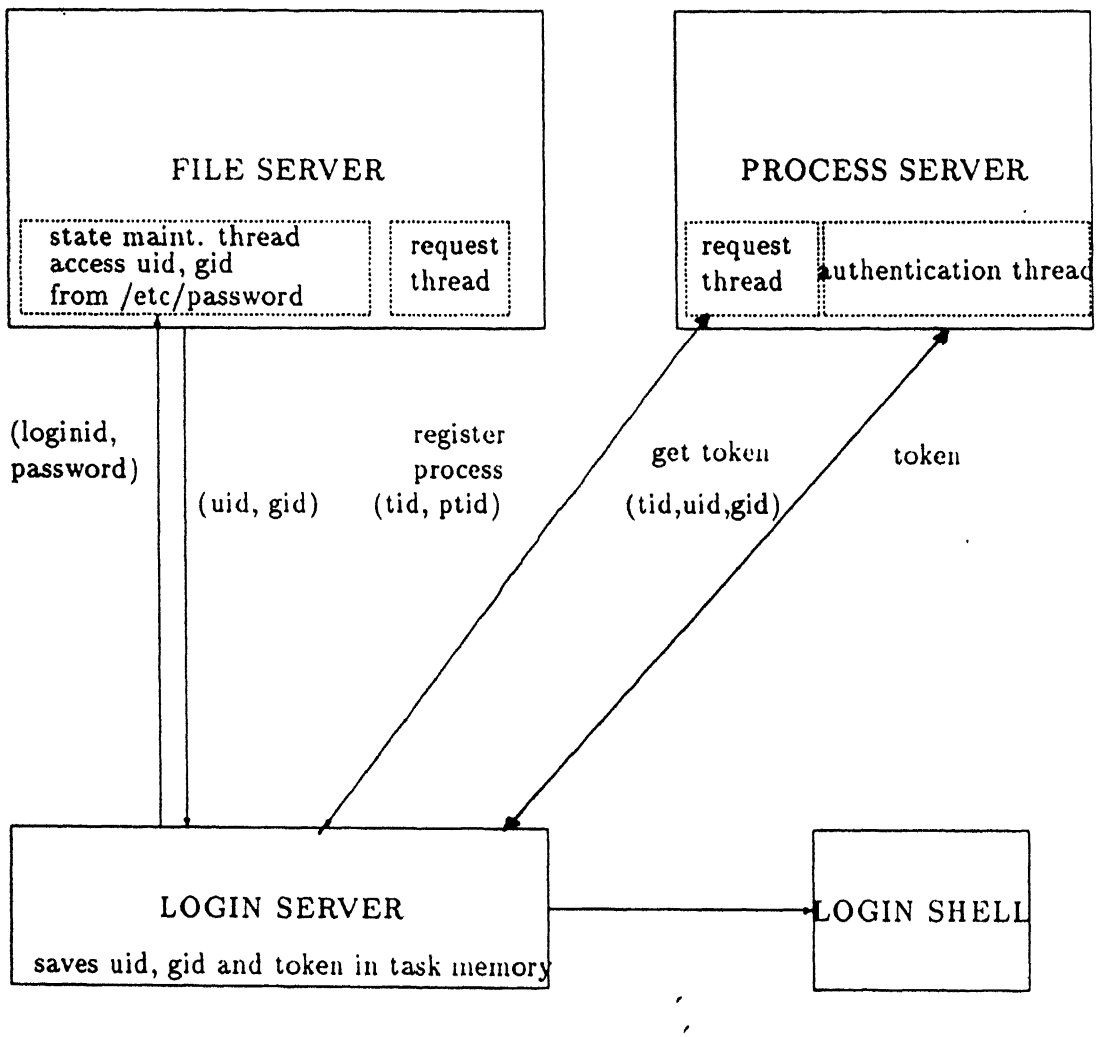


Figure 4.4: Initialisation of credentials

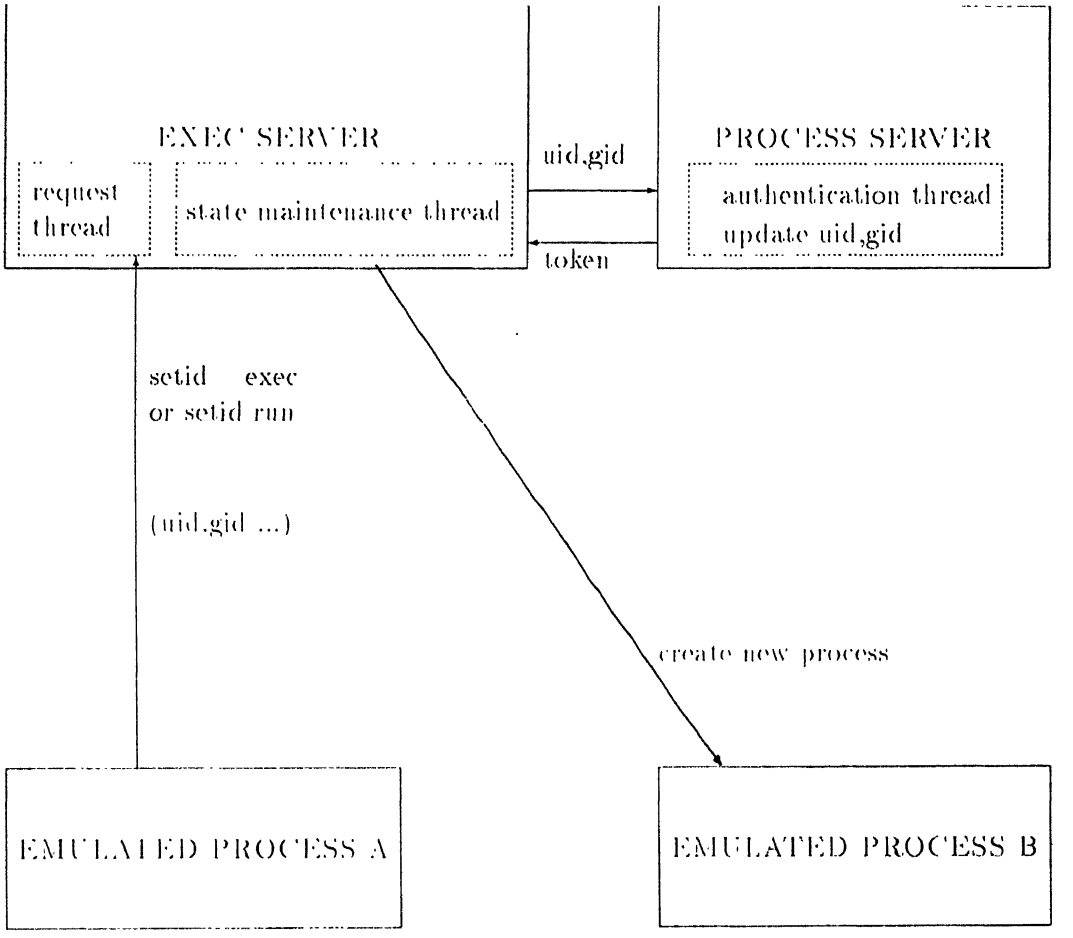


Figure 4.5: Update of credentials

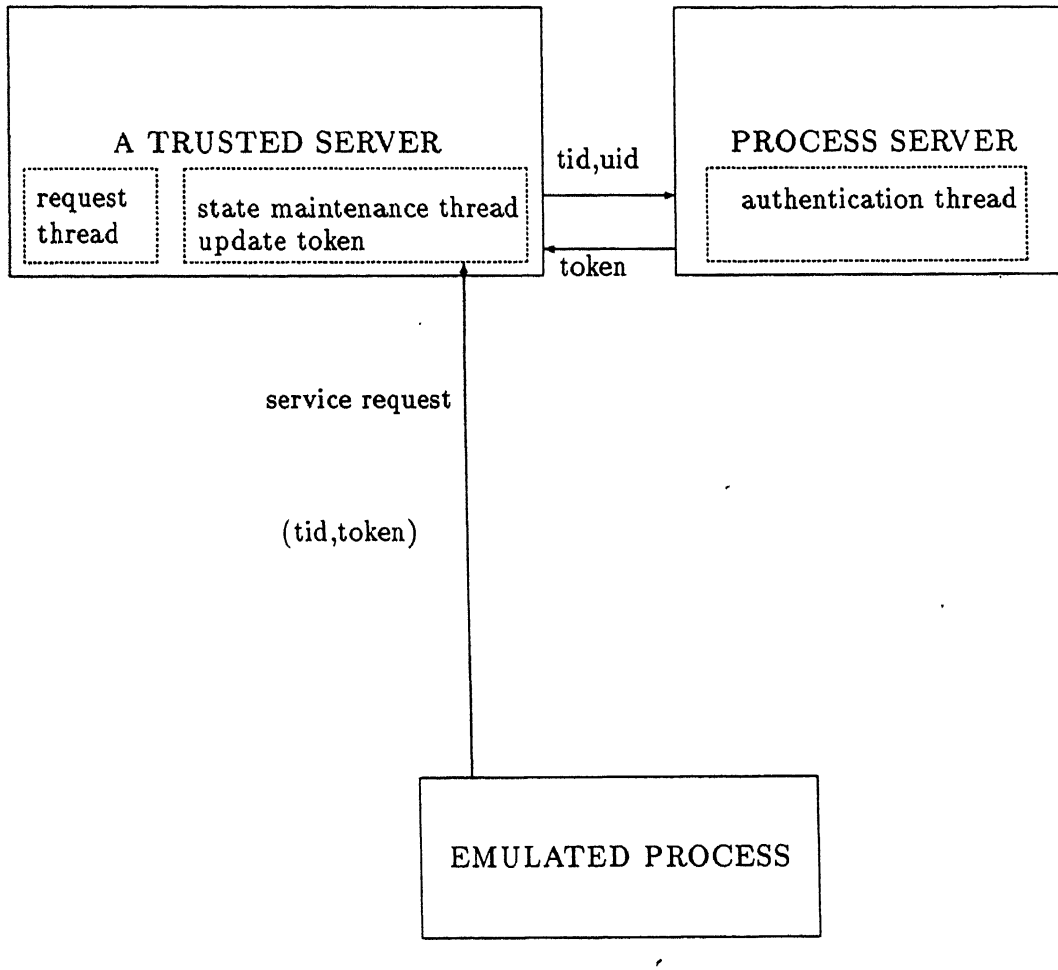


Figure 4.6: Authentication of requests

discussed below.

- *get_creds*: This is called by the login server, which sends the user_id and group_id associated with the new user process. A new token is assigned by the authentication server and relevant entries are made in the user_id and group_id fields in the signal table.
- *check_creds*: This call is made by the other trusted servers when they want to authenticate the user request. The token associated with the user is returned to the servers. The servers make use of the token to check the validity of the requests made to them.
- *update_creds*: A new token is assigned to the user and the user_id and group_id associated with it is updated to reflect its change in credential. This call is made during setid system call or by the exec server during a setid exec.

4.6 Conclusion

The process server along with the emulation library provides a complete implementation of the UNIX signalling facility. The process control information within the server is used by the emulation library to implement the *wait* system call. The efficacy of the signalling mechanism is proved by the easy implementation of a simple login shell. The authentication server forms an adequate support for the protection mechanism and guarantees the security of our system .

Chapter 5

Program Execution in PAWAN

5.1 Introduction.

In this chapter we discuss the implementation of the *execve* call as a emulation library routine. We have provided a trusted server, the Exec Server to handle `setid` exec, which requires updation of the associated privileges. The implementation of the other system calls used for process control is given in chapter 6. In the next section we give a brief review of process execution in UNIX. In sec 5.3 we make an examination of such implementations in other systems. In the remaining part of the chapter we describe the PAWAN implementation in detail.

5.2 Process Execution in UNIX

A process is a program in execution, which is assigned system resources such as memory and the underlying CPU. A UNIX process operates in either user mode or kernel mode. In user mode, a process executes application code with the machine in a nonprivileged protection mode. When a process requests services from the operating system with a system call, it switches into the machine's privileged protection mode and operates in kernel mode [Bach86], [Leff89]. The operation of executing a new process is carried out by creating a skeletal process and embedding the required state into it. This involves allocating and initialising operating system state and other system resources, loading the new program in memory and making it executable. In UNIX, every process except process 0 (special process created 'by hand' when the system boots) is created when another process executes the *fork* system call, which replicates the calling process. Then the *execve* system call is executed which overlays the current image by a new executable program.

5.2.1 UNIX Implementation of *Execve*

The *execve* system call overlays the memory space of a process with a copy of the specified executable file. The file is either an executable object file, or a file of data for an interpreter. The contents of the user-level context that existed before the *execve* call are no longer accessible afterward except for the call parameters, which the kernel copies from the old address space to the new address space. The parameters consist of the name of the new program, the argument vector for the executable program and environment variables.

The executable object file consists of an identifying header, followed by pages of data representing the initial program (text) and initialised data pages. The header contains an exec structure, which describes the sizes of various segments, loader format, symbol table address and other information useful for debugging. The magic number field of the header is a short integer, which identifies the file as a load module and enables the kernel to distinguish its run-time characteristics. The magic number specifying the loader format can be one of the following.

- OMAGIC
  A text executable image which is not to be write protected, with a contiguous data segment.
- NMAGIC
  A write protected text executable image. The data segment begins at the first segment boundary following the text segment.
- ZMAGIC
  A page aligned text executable image. The data segment begins at the first segment boundary following the text segment. The text and data sizes are multiples of page size. File is demand paged, read only.

An interpreter file begins with a line of the form '#! interpreter [arg]'. When the interpreter file is *execve*'d, the system executes the specified interpreter. If the optional argument is defined it becomes the first argument and the name of the original *execve*'d file becomes the next argument. The original arguments are shifted over to become the subsequent arguments.

There can be no return from a successful *execve* since the calling core image is lost. Descriptors open in the calling process remain open in the new process, except for those for which the close-on-exec flag is set. Ignored signals remain ignored, but the signals that are caught are reset to their default values. Blocked signals remain blocked regardless of changes to the signal action.

The kernel takes special action for `setid` programs and for process tracing. When the kernel execve's a setid program. the effective User ID (or group ID) fields in the process table and uarea are set to the owner ID of the file.

5.3 Critical Review of Other Implementations.

The experiences with UNIX led to the evolution of certain alternate strategies to circumvent the factors producing inefficiency in the UNIX approach to process creation and execution.

A conclusion which was swiftly drawn was that a lot of waste was inevitable when a fork was followed by an execve with very little code in between them. The entire address space of the parent is duplicated at the time of *fork*, and thrown away when *execve* is invoked. The performance further deteriorates if parts of the parent's address space are on secondary storage, since they have to brought back in main memory. Many different schemes have been suggested to circumvent this.

UNIX system V duplicates address space by means of copy-on-write mechanism which defers physical copying of a page until it is modified. This scheme still entails allocation and initialisation of page maps, which is redundant if the pages are not going to be accessed.

BSD4.3 provides *vfork* for operations in which the parent and child does not run concurrently. Instead of copying the address space for the child, the parent simply passes its address space to the child and suspends itself. The child process returns from the vfork system call with the parent still suspended. When the child does an exit or an execve the address space is passed back to the parent process. Although extremely efficient, vfork has peculiar semantics and is generally considered to be an architectural blemish. The architectural quirk of vfork is that the child process may modify the contents and even the size of the parent's address space while the child has control [Leff89].

LOCUS [Walk83] provides *run* along with *fork* and *execve*. The run system call is actually fork with the definite knowledge that execve is going to follow. The forking is done in such a manner that the address space is constructed according to the requirements of execve rather than duplicating it from the parent.

System V and the BSD4.3 solutions still provide only fork to create a new process, and try to improve its efficiency. LOCUS defines another way to create a new process, but it lacks in generality. In MACH we provide alternate means to create process with known requirements. The parent can read/write a child's address space and can mark specific regions of address space for read-write sharing or copy-on-write sharing with the child [Acce86], [Golu87]. We can thus achieve the desired functionality efficiently.

5.4 PAWAN Approach to Process Execution.

PAWAN provides *fork* (refer chapter 6) for process creation and *execve* and *run* system calls for process execution. These are implemented as emulation library routines. We have defined a trusted server, the Exec Server to handle

setid execve and run. The Exec Server runs with root privileges at startup time. The execve and run library routines send a request to the exec server when it comes across a setid execve or run.

The basic algorithm followed by execve and run is what was proposed by Ashish Singhai [Ashi 92]. The only difference is in the file related calls made during execve and run. In this case we interact with the file server to access the necessary information.

5.4.1 PAWAN Implementation of *Execve*

We have implemented the *execve* system call as a emulation library routine, which sends a special request to the Exec server asking it to take over, when it comes across a `setid` execve. The routine must be able to write the code and data segments of the executable file into the memory [Ashi92]. This is not possible if the code of execve itself resides in the code segment. Since we did not want to modify the kernel to accommodate these calls, we decided upon the following strategy to load the file. The file loading part of the execve code (`exec_code`) resides in a fixed region of memory with its stack lying adjacent to it. The memory map of the user process is shown in [fig. 5.1]. The `exec_code` is coded in a reentrant way and separately compiled. The exec_code will already be present in the process memory. This might have been set up by its parent at the time of a *run* or inherited from the parent during a *fork* emulation library call. After the initial processing we do a non local 'goto' to the exec_code. The initial operations consist of checking the mode and access permissions, resetting its state in various servers and setting up the process stack [Ashi 92].

The exec_code deallocates the current user stack and copies the process stack into the correct location. The executable file is then loaded as specified by the magic number. It now jumps to the user program and starts execution.

5.4.2 PAWAN Implementation of *Run*

The *run* system call has also been implemented as an emulation library routine. When a process makes a *run* call, a child process is created and made to execute the specified file. We utilize the *task_create* kernel call (with memory inheritance deactivated) provided by the MACH kernel. The parent process memory is not replicated in the child since it is thrown away when we load the executable file into the child's memory space. The parent task requests all the trusted servers to replicate its state for the child task. The information replicated in the various servers is shown below

- PROCESS SERVER
  A *task_create_msg* call (refer chapter 4) is made.

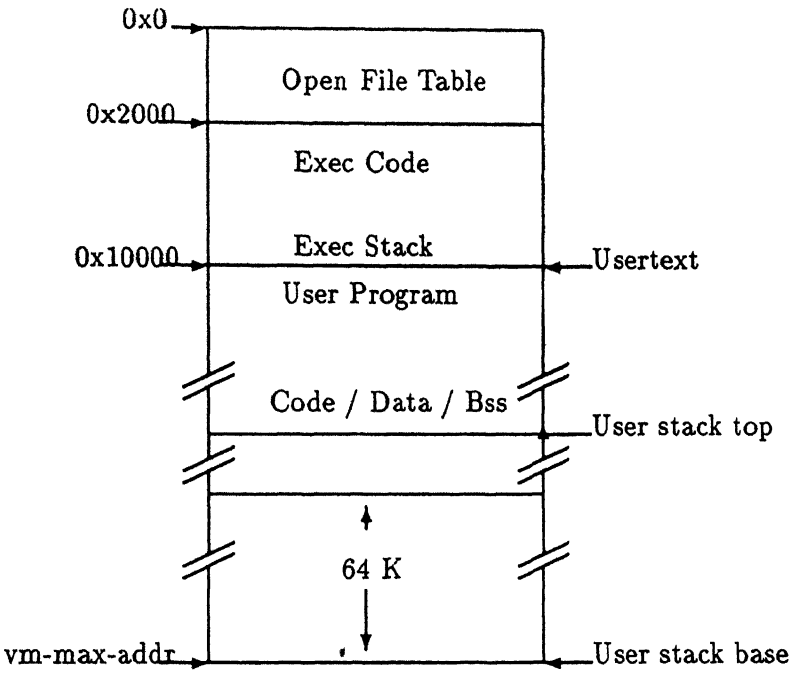


Figure 5.1: The Memory Map of A PAWAN Process

- FILE SERVER
  A new entry is made for the child and the working directory, root directory, file mode mask values etc. are replicated in the child, and the reference counts of shared objects (open files, current directory etc.) are increased [Bair93],[Rao92].

- TTY SERVER
  A uarea entry is created for the child and the process group and control terminal is inherited from the parent [Gopa92].

- ENV SERVER
  We have the parent and child task sharing the same environment instead of making a specific copy of the environment for the child [Baru92].

Now the permission and mode of the executable file is checked, if the setuser ID or setgroup ID mode of the file is set a request is sent to the Exec Server and the parent process continues execution.

The exec_code is setup by the calling process and loaded into the corresponding location in the child's memory space. This is possible since the parent task is allowed to make kernel calls on behalf of the child. The user stack for the child is also assembled by the parent process and loaded into the corresponding location. The child is made to start executing the exec_code and the parent returns to its calling point. Since the user stack is already

set up, the exec_code just loads the specified file in the format stipulated by the magic number and starts executing the program.

5.4.3 *Execve* and *Run* semantics

The signal related variables are stored at fixed locations in the user address space. During an execve we deallocate the signal thread and port (if the calling process handled signals) and inform the process server. The signals that are caught as well as ignored are reset to their default values. In BSD4.3 ignored signals remain ignored, while caught signals are reset. We decided against following these semantics in PAWAN since it involves setting up a new signal thread to accept the signal messages from the process server. This was thought to be wasteful since a new program hardly ever makes assumptions about its signal state.

The open file table of a process is loaded in a fixed location in the user memory. After the *execve* emulation library routine this table remains in the user space. Thus the new program is able to access the file opened by the process before the execve, unless they have been closed explicitly before invoking execve. The child process created by the run routine receives a copy of the open file table from the parent. So the child is also permitted to access the files which were opened by the parent and not closed before invoking run. To summarize, we have the loader as part of the user space, and a user level implementation of the execve call. Vfork has been chucked out, and run has been implemented.

5.5 Exec Server

We had to provide a server implementation of the setid calls, which involves updating the privileges associated with the emulated process, since privileged calls can only be made by a trusted server. The security issues have been described in chapter 4. In fact, the process server itself can take care of the setid exec, rather than setting up another server.

A security hazard that is evident at this point, is the ability of the parent to overlay the code in the child's virtual memory even after the privileges have been updated. This is possible since the parent has access to the child's kernel port and can make kernel calls on behalf of the child.

The Exec Server exports the *setid_exec* call which is invoked by the *run* and *execve* library routines when it detects that the setuser ID or setgroup ID flag of the executable file is set. The arguments to this call will be the task (whose privileges are to be updated), the new values used to set the effective group or user ID and the arguments to the execve or run library call. Separate calls need not be provided for execve and run since only the process which is to run the program is relevant now. The run routine passes

on the child task created by it, while the execve routines specifies the calling process itself.

5.5.1 Exec server Implementation of *setid_exec*

At the receipt of a setid_exec request, the major task performed by the Exec Server is the updation of the associated privileges. The entries in all the trusted servers have to be updated to reflect the change in credentials.

We have defined the following strategy to circumvent this security hazard described in the above section. The Exec Server creates a new task and the privileges are assigned to this task and the old child task is terminated. We overwrite the old child task's server states rather than creating new entries for them. This is necessary since otherwise the child will seem to have died from the parent's perspective. The server based implementation degrades the performance, but this is acceptable since frequency of such calls is comparitively less.

5.5.2 Server State Updating

Since the exec server is itself a trusted server, it makes privileged calls to each of the servers requesting it to change the privileges associated with the task. The server state updating in each trusted server is described below. The kernel port entries in all the servers are updated.

- SIGNAL SERVER
  The Exec server calls *update_creds* (refer chapter 4).

- FILE SERVER
  The table entries are altered to reflect the change in credentials [Rao92], [Bair93].

- TTY SERVER
  The uarea entries are updated with the new values for user ID and group ID [Gopa92].

Once this is done the Exec server sets up the exec_code (and its stack) and processes the argument vector and environment variables to construct the user stack for the new task . A thread a created and it starts executing the exec_code. Now the new process is on its own and once the executable file is loaded it starts executing the specified program.

5.6 Conclusion

The shortcomings of this scheme is that the open file table and exec_code are user writable. The operation of signals, and execve will be impeded if the code is accidently overwritten [Ashi 92]. The positive aspect to the user level implementation of the execve and run routines is that it promotes flexibility. The implementation can be easily adapted to interface with more servers in the case of augmentation of prevailing services with additional servers. Different loaders and object file formats can be implemented.

Chapter 6

PAWAN Emulation Library

6.1 Introduction

The design of the PAWAN emulation services for UNIX has been described in chapter 3. We rely on the emulation library to act as an interface translator, enabling UNIX services in a completely transparent manner. We can, in fact construct a wide range of other user higher level systems on top of the PAWAN servers by providing the corresponding services via emulation library routines [Juli92]. Our implementation allows us to execute UNIX programs as if they are running under a native implementation of the UNIX system. The emulation library associated with each emulated process, intercepts the system calls issued by that process and redirects them to appropriate emulation services. A considerable portion of the process state and the emulation functions are managed locally by the emulation library.

6.2 UNIX Process Control

UNIX implements four major system calls for process control : fork, exec, wait, and exit. The *fork* system call creates a new process, the *exit* call terminates process execution, and the *wait* call allows a parent process to synchronise its execution with the exit of a child process. Signals inform processes of asynchronous events. The *exec* system call allows a process to invoke a new program, overlaying its address space with the executable image of a file [Leff89], [Bach]. We provide these calls in PAWAN through emulation library routines. This enables us to provide an environment in which users can continue using programs that use the UNIX system call interface. The emulation library interacts with the PAWAN Process Server to implement the UNIX signalling interface. The Process Server implementation has been described in chapter 4. In section 6.2 we enumerate the emulation routines handling the signalling related calls. The fork, exit and

```
signal_init()
{
 mach_port_t psport, excport, mykp;

 get_process_server_port(&psport); /*accessed from the env server*/
 task_create_msg(psport,mykp,0); /*register task */
 task_get_exception_port(mykp,&excport); /*get task exception port*/
 set_sigport(psport,mykp,excport); /*set exception port as sigport*/
 signal_thread_start(excport);
}
```

Figure 6.1: Algorithm of the initialization routine

wait routines are presented in subsequent sections. PAWAN handling of program execution is explained in chapter 5.

6.3 Emulation Routines Related to Signalling.

Signal related variables like current mask, pending signals etc are stored at fixed locations in the user address space, The management (initialisation and updation) and protection of these variables thus falls on the emulation library. We have implemented a locking mechanism to protect these data against corruption during updation. UNIX signalling is implemented using message passing in PAWAN. Since PAWAN allows multiple threads of execution within a task we don't need to stop the user thread (the main thread of execution, which with the task is analogous to a UNIX process). We can have a separate thread in each task waiting for signals. However, we have opted for having such a thread only for those tasks interested in handling signals.

6.3.1 Signal State Initialisation.

When a user program makes its first signal related call (e.g. sigvec signal etc), the emulation library associated with each process creates a signal thread which waits for messages on a per task signal port (the exception port of the task by default). The algorithm of the initialisation routine is shown in fig. 6.1.

6.3.2 Signal Posting and Delivery

The signal thread waits for signal messages from the process server and posts the signal at the receipt of a message. If the signal action associated with the signal is SIG_IGN, nothing is done. If SIG_DFL is specified the

threads set a flag in the reply message and the default action is taken by the process server. If neither of this holds true, the signal is added to the list of pending signals for the task. The signal thread executes the *issig* each time a message is received on its signal port. Within the issig function, the thread removes the masked and ignored signals from the list of pending signals. Now a signal is selected from the list of pending signals and the current signal is set to that value. If *issig* returns 'true', a subroutine is invoked to deliver the signals. Since the signal mask and default action associated with signal could have changed in the interval between the invocation, we do some validity checking. The signal is blocked from further occurrence and a new signal mask is installed for the duration of the process' signal handler. This mask is formed by taking the current signal mask, adding the signal to be delivered, and ORing in the signal mask associated with the handler to be invoked. If the SV_SOMASK flag is set, this denotes that a sigpause call has been made and so the signal mask has to be set to the old mask value after the specified action is executed. The emulated process is initially created with a single thread of execution (we refer to it as the UNIX thread). If the task makes any signal related calls, implying that it wants to handle signals, a signal thread is created which waits on the signal port (by default, the exception port) for signal messages from the process server. When a signal message arrives on the signal port, the signal thread suspends the execution of the UNIX thread (to emulate the UNIX process which has only one thread of execution), and executes the affiliated action. If the interrupt bit is set, we invoke the *thread_abort* kernel call to abort any system call that is in process. The main thread of execution is resumed when the handler routine returns.

6.3.3 Signal Related Calls

We have provided the BSD4.3 sigblock and sigsetmask system calls to modify the set of masked signals for the process. The sigblock call adds to the set of masked signals, whereas the sigsetmask call replaces the set of masked signals. We have also implemented the sigpause call, which relinquishes the processor until it receives a signal and the sigvec and signal system calls to specify an action for the signal. The sigstack system call of BSD4.3 allows users to define an alternate stack on which the signals are to be processed. Since signals in PAWAN are delivered by the signal thread, the signals are automatically processed by an alternate stack. Thus it was not deemed necessary to provide the sigstack system call.

6.4 Fork Emulation Library Routine

In UNIX, new processes are created using the fork system call. On return from the fork system call, the two processes have identical copies of their

user level context, except for the return value. In the parent process, the return value is the child process_ID; in the child process, return value is 0 [Leff89], [Bach86]. To implement this call in PAWAN, the parent emulated process which is normally in contact with many servers must arrange for the child process to inherit access to all the servers along with a consistent user state. At the invocation of the fork library routine, the following sequence of actions are taken. The task_create kernel call is executed with the inheritance flag set to TRUE, thus creating a child task with all memory shared copy-on-write. Now messages are sent to all the servers, specifying the parent and child tasks, and asking them to duplicate the server state of the parent in the child (similar to the requests made at the time of 'run' : refer chapter 5) . The process server assigns a new process_ID to the child and sets its parent process_ID correctly. A thread is created for the child task. We need to duplicate the state of the parent task's calling thread in the child thread so as to provide fork semantics. This is implemented in the following manner. The MACH kernel provides the call, thread_get_state to access the state of the thread, but this can be invoked to access the state of the calling thread itself. Rather than modifying the kernel by adding one more call, we made use of the following strategy. A new thread is created for the parent task, and this is made to access the state of the calling thread and duplicate it in the child thread. Once this is done the child thread is started. The thread created temporarily for this purpose is now terminated. The parent task accesses the child process_ID from the process server and returns. The child thread starts executing from this point and returns to the calling point with a value 0.

6.5 The Exit Emulation Library Routine.

Processes on a UNIX system terminate by executing the exit system call. There is provision to specify the exit status, which is returned to the parent process for its examination [Bach86], [Leff89]. We have an exit kernel call, which terminates the task performing the emulated process. We have implemented the exitp emulation library routine which informs all the servers about the termination of the emulated process and then invokes the exit kernel call. On receiving a termination message the following actions are performed by the servers.

- PROCESS SERVER : Once the request is authenticated (only the task itself or its ancestor is allowed to invoke the task_terminate_call), we send a signal to the parent task informing the change in status. The child exit status and process_ID is saved for subsequent access by the parent during a wait. The process table entry is now deallocated.

- FILE SERVER : All the open file descriptors are closed and the relevant entries are deallocated.

PROCESS MANAGEMENT COSTS		
OPERATION	BSDV1.6	PAWAN
FORK	16 ms	398 ms
EXECVE	66 ms	840 ms
RUN	-	1.6 secs

Figure 6.2: Process Management Costs

- TTY SERVER : All the open terminals associated with the emulated process is closed, and all the relevant structures are freed.

Similarly the environment server entries are also deallocated.

6.6 The Wait Emulation Library Routine

A UNIX process can synchronise its execution with the termination of a child process by invoking the wait system call. The kernel searches for a zombie child of the process and, if there are no children returns an error code. If it finds a zombie child, it extracts the process_ID and the status specified by the child's exit call, and returns these values [Bach86]. To implement this in PAWAN, we make use of the get_cexit_status call exported by the process server. The process server contains information about the process_ID and termination status (the status bit specified by the exit routine, or the signal which terminated or stopped the process). When a get_cexit_status call is received on its public port, the server searches for child tasks of the calling task, it returns error if none are found. If there is any zombie child, it returns the required information. If there are no zombie children this information is sent to the routine, which suspends the emulated process (this remains suspended till the receipt of a signal).

6.7 Conclusion

A small amount of performance testing has been done for the process control calls (Fig. 6.1). A major factor contributing to the cost of these operations , is the updating of process state in the servers during the *fork* and *run* calls. A possible modification would consist of allowing the child to automatically inherit the parent process state in all the servers. This however will result in the servers losing their ability to differentiate betwen the child and parent [Juli92]. We have followed this method in case of the Env server [Gopa92].

But the Process server needs to differentiate between parent and child processes for implementing *kill*. Efficiency was not initially our driving concern. Now we have to identify the possible bottlenecks, and take corrective action.

The file system calls (e.g. open, close, read etc.) has been implemented by the file server[Bair93] and tty server [Gopa92]. So the emulation library just reroutes these calls to the appropriate servers. The C library functions like printf, putc, have been easily ported without any major changes.

Chapter 7

Conclusion

The work described in this, and the companion thesis, exhibits the successful implementation of an MACH-based operating system which emulates UNIX. As stressed in the earlier chapters, the major motivation and goal behind this thesis has been to construct an operating system as a set of system servers which sit on top of a minimal Micro-kernel and support the implementation of functions that are specific to a particular operating system. This approach promises to meet system builders' need for a sophisticated operating system development environment that can cope with growing complexity, new architectures, and changing market conditions. And the fact that PAWAN has met its design goals further substantiates the claim that this approach is here to stay.

The very idea of microkernel based operating systems is rooted in the belief that in order to make way for configurability, potential inefficiency in certain cases is not unwelcome. And in our case, efficiency has never been a major goal. As discussed in the previous chapter, the efficiency issues have to be treated seriously to make PAWAN a real success. Therefore, tuning up of PAWAN to achieve better efficiency may be a possible extension to this project.

Any research in operating systems, more particularly in distributed systems, is presently influenced to a large extent by attempts to stay compatible with UNIX. MACH is no exception to this philosophy. Although PAWAN successfully emulates UNIX, it is not yet a full fledged operating system. Enrichment of the emulation library and porting of standard utilities is another possible extension to this project although this might turn out to be a never ending process. The most important among them being an editor and the gcc compiler. At present, programs have to be edited and compiled on the SUN workstations. It may be pointed out here that porting of the Bourne Again Shell was one of the goals set in the beginning. But it never materialized owing to lack of time. The login shell which is started by the TTY Server may be written as an independent server.

The potential benefits of micro-kernel support for real-time systems has been discussed earlier. This area is open for exploration and PAWAN can serve as a platform for research in this direction.

We have not provided any sort of process accounting in our emulation system. The process state is distributed between the different servers and the emulation library. The Process server can be extended to provide this facility. Just a few C library functions have been implemented. The emulation library has to be further extended to handle the other UNIX system calls.

Bibliography

[Abro89] Abrossimov,V., Rozier,M., Gien,M., *Virtual Memory Management in CHORUS*, LNCS, Vol 433, Springer-Verlag, 1989.

[Acce86] Accetta,M., Baron,R., Golub,D., Rashid,R., Tevanian,A., and Young,M., *MACH: A New Kernel Foundation for UNIX Development*, Technical Report, School of Computer Science, Carnegie Mellon University, August 1986.

[Ashi92] Ashish,S., *PAWAN: A MACH based UNIX system(I)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Bach86] Bach,M.J., *The Design of the UNIX Operating System*, Prentice-Hall Inc., Engelwood Cliffs, May 1986.

[Bair93] Bairagi,D., *UNIX Emulation Services in PAWAN(II)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1993.

[Baro88] Baron,R.V., Black,D., Bolosky,W., Chew,J., Draves,R.P., Golub,D.B., Rashid,R.F., Tevanian,A.Jr., and Young,M.W., *MACH Kernel Interface Manual*, Online MACH Documents (unpublished), School of Computer Science, Carnegie Mellon University, October 1988.

[Baru92] Baruah,M., *PAWAN : A MACH based UNIX System(III)*, M. Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Blac89] Black,D.L., *Mach Interface Proposals - Priorities, Handoff, Wiring*, August 1989. Unpublished mánuscript from the School of Computer Science, Carnegie Mellon University.

[Bolo87] Bolosky,W., Rashid,R., Tevanian,A.Jr., Young,M., Golub,D., Baron,R., Black,D., and Chew,J., *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures*, Technical Report CMU-CS-87-140, School of Computer Science, Carnegie Mellon University, July 1987.

[Cher84] Cheriton,D.R., *The V-Kernel: A Software Base for Distributed Systems*, IEEE Software, Vol 1 no.2, pp. 77-107.

[Cher88] Cheriton,D.R., *The V Distributed System*, Communications of the ACM, 31(3), March 1988.

[Coop88] Cooper,E.C., and Draves,R.P., *C Threads*, Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February 1988.

[Coul88] Couloris,G.F., Dollimore,J., *Distributed Systems : Concepts and Design*, Addison-Wesley, July 1988.

[Dean90] Dean,R., Golub,D., Forin,A., Rashid,R., *UNIX as an Application Program*, Proceedings of the 1990 Summer Usenix, USENIX, June 1990.

[Drav89] Draves,R.P., Jones,M.B., and Thompson,M.R., *MIG: The MACH Interface Generator*, Online MACH Documents (unpublished), School of Computer Science, Carnegie Mellon University, July 1989.

[Gien91] Gien,M., *Next Generation Operating System Architecture*, LNCS, Vol. 563, July 1991.

[Gold89] Goldstien,I., *Introduction to th MACH Development workshop*, Proceedings of the Workshop on MACH Development, Boston, 1989.

[Golu87] Golub,D.B., Tevanian,A.Jr., Rashid,R., Black,D.L., Cooper,E., and Young,M.W., *MACH Threads and the UNIX Kernel: The Battle for Control*, Technical Report CMU-CS-87-149, School of Computer Science, Carnegie Mellon University, August 1987.

[Gopa92] Gopal,B., *PAWAN: A MACH based UNIX System(II)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Jone86] Jones,M.B., Rashid,R.T., *MACH and Matchmaker : Kernel and Language Support for Object Oriented Distributed systems*, ACM Sigplan Notices, Vol. 21, no.11, pp. 67-77.

[Juli92] Julin,D.P., Chew,J.J., Stevenson,J.M., Guedes,P., Neves,P., Roy,P., *Generalised Emulation Services for MACH3.0: Overview, Experiences and Current status*, MACH Symposium, USENIX Association.

[Leff89] Leffler,S.J., McKusick,M.J., Karels,M.J., Quarterman,J.S., *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley Publishing Co., May 1989.

[Mull85] Mullender,S.J., Tanenbaum,A.S., *A distributed File Server Based on Optimistic Concurrency Control*, Proceedings of the Tenth ACM Symposium on Operating Systems, Dec. 1985.

[Pete90] Peterson,L.M., Hutchinson,N.C., O'Malley,S.M., and Rao,H.,C., *The X-kernel: A Platform for Accessing Internet Resources*, IEEE Computer, 23(5), May 1990.

[Rao92] Rao,P.V., *PAWAN : A MACH based UNIX System(IV)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Rash86] Rashid,R.F., *Threads of a New System*, UNIX Review August 1986.

[Rash87] Rashid,R.F., *Designs for Parallel Architectures*, UNIX Review, April 1987.

[Rene89] Renesser,R.V., Tanenbaum,A.S., *The Evolution of a Distributed Operating System*, LNCS, Vol 433, Springer-Verlag.

[Tane90] Tanenbaum et al., *Amoeba - A Distributed Operating System for the 1990's*, IEEE Computer, Vol. 18 No. 2, 1990.

[Teva87] Tevanian,A.Jr., and Rashid,R., *MACH: A Basis for Future UNIX Development*, Technical Report CMU-CS-87-139, School of Computer Science, Carnegie Mellon University, Pittsburgh, June 1987.

[Thom87] Thompson,M.R., Tevanian,A.Jr., Rashid,R., Young,M.W., Golub,D.B., Bolosky,W., and Sanzi,R., *A UNIX Interface for Shared Memory and Memory Mapped Files Under MACH*, Technical Report, School of Computer Science, Carnegie Mellon University, Pittsburgh, July 1987.

[Walk83] Walker et al., *The LOCUS Distributed Operating System*, Proceedings of the Ninth Symposium on Operating Systems Principles, October 1983.

[Youn87] Young,M., Tevanian,A.Jr., Rashid,R., Golub,D., Eppinger,J., Chew,J., Bolosky,W., Black,D., and Baron,R., *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System*, Proceedings of the Eleventh ACM Symposium on Operating System Principles, Austin, Texas 1987.